2019_Clamons-Murray_Modeling-CRISPRa-Bottlenecking_figure_generation

July 29, 2019

In [1]:
```
import math
import copy
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import matplotlib as mpl
import scipy

# numdifftools required for sensitivity analysis
import numdifftools as nd


mpl.rc('xtick', labelsize=18)
mpl.rc('ytick', labelsize=18)
mpl.rc('axes', linewidth=1.75, titlesize=48)
plt.gcf().subplots_adjust(bottom=0.15)

import IPython.display


'''
This block is only for formatting. If you do not have the
seaborn package (and don't want to get it), you can safely comment
out or delete everything from...
'''
########
# HERE #
########
import seaborn as sns
rc={'lines.linewidth': 2, 'axes.labelsize': 14, 'axes.titlesize': 14}
sns.set(rc=rc)
sns.set(style = "ticks")

sns.set_context("talk", font_scale=1, rc={"lines.linewidth": 2.0, 'lines.ma
sns.set_style("ticks")
```

```
        sns.set_style({"xtick.direction": "in","ytick.direction": "in"})
        mpl.rc('text', usetex=False)
        tw = 1.5
        sns.set_style({"xtick.major.size": 6, "ytick.major.size": 6,
                       "xtick.minor.size": 4, "ytick.minor.size": 4,
                       'axes.labelsize': 48,
                       'xtick.major.width': tw, 'xtick.minor.width': tw,
                       'ytick.major.width': tw, 'ytick.minor.width': tw,
                       'axes.titlesize': 48})

        mpl.rc('xtick', labelsize=18)
        mpl.rc('ytick', labelsize=18)
        mpl.rc('axes', linewidth=1.75)
        plt.gcf().subplots_adjust(bottom=0.15)
        sns.set_style({'axes.labelsize': 24})

        ###########
        # TO HERE #
        ###########

        %matplotlib inline
```

In [2]:
```
        %%javascript
        MathJax.Hub.Config({
            TeX: { equationNumbers: { autoNumber: "AMS" } }
        });
```

```
<IPython.core.display.Javascript object>
```

1 Introduction

This notebook reproduces the plots used in "Modeling predicts that CRISPR-based activators, unlike CRISPR-based repressors, scale well with increasing gRNA competition and dCas9 bottlenecking", by Samuel Clamons and Richard Murray. It also includes an explanation of the model used to generate those results.

In brief, we compare the performance of CRISPRi (according to Zhang & Voigt 2018, https://doi.org/10.1093/nar/gky884) and CRISPRa (according to Dong & Zalatan 2018, https://doi.org/10.1038/s41467-018-04901-6) as each experiences increasing competition from off-target gRNAs. I predict that activators scale better, using the model from Zhang & Voight, which was adopted from Chen, Qian, & Del Vecchio 2018 (https://www.biorxiv.org/content/biorxiv/early/2018/02/14/266015.full.pdf). We got some assistance understanding the Shea-Ackers model (cited by Zhang & Voigt in equation (11)) here: http://vcp.med.harvard.edu/papers/SB200-6.pdf

2 Utility Functions

In [3]:
```
# Generate some useful colormaps with nice, colorblind-friendly
# colors, in both discrete and continuous flavors.
def make_fade_colormap(r, g, b, name):
    '''
    Make a new continuous colormap fading from white to a
    color (r, g, b).
    '''
    cdict = {"red": ((0.0, 1.0, 1.0), (1.0, r/256, r/256)),
             "green": ((0.0, 1.0, 1.0), (1.0, g/256, g/256)),
             "blue": ((0.0, 1.0, 1.0), (1.0, b/256, b/256))}
    return colors.LinearSegmentedColormap(name, cdict)

def discretize_fade_colormap(cmap, min_c, max_c, N):
    '''
    Convert a range [min_c, max_c] of a continuous colormap cmap
    into a discrete colormap with N values.

    Code modified from https://scipy-cookbook.readthedocs.io/
    items/Matplotlib_ColormapTransformations.html
    '''
    colors_i = np.concatenate((np.linspace(min_c, max_c, N), (0.,0.,0.,0.)))
    colors_rgba = cmap(colors_i)
    indices = np.linspace(0,1, N+1)
    cdict = {}
    for ki, key in enumerate(('red','green','blue')):
        cdict[key] = [(indices[i], colors_rgba[i-1,ki], colors_rgba[i,ki])
                      for i in range(N+1)]
    # Return colormap object.
    return colors.LinearSegmentedColormap(None, cdict, 1024)

green_fade_colormap = make_fade_colormap(27, 158, 119, "GreenFade")
orange_fade_colormap = make_fade_colormap(217, 95, 2, "OrangeFade")
purple_fade_colormap = make_fade_colormap(117, 112, 179, "PurpleFade")
```

3 The Model

To get the fold-change of a repressing dCas/guide pair, treat the guide-bound dCas as a Shea-Ackers-like repressor of cooperativity 1 and a single binding site. We can then write down the average transcription rate of a target promoter as an average of rates r_f and r_C of transcription from free and dCas-bound promoter, respectively, weighted by the probability of finding the promoter in each of those states:

$$r = r_f P + r_C K C_{s1} P \tag{1}$$

- r: Average transcription rate.
- r_f: Transcription rate from free promoter.

- r_C: Transcription rate from bound promoter.
- P: Concentration of free promoter.
- K: *Association* constant of dCas:g1 binding to promoter.
- C_{s1}: Concentration of dCas:g1 complex.
- β: Degradation rate of gRNA / association constant of gRNA binding to dCas.
- α: Transcription rate of gRNA.

When the repressor isn't present, this simplifies to $r = r_f P$; therefore, the fold-change caused by the repressor is:

$$RFC = \frac{r_f P_{tot}}{r_f P} = \frac{r_f (P + K C_{s1} P)}{r_f P} = 1 + K C_{s1} \tag{2}$$

Using some steady-state assumptions and dynamics of C and C_{s1} and s_1 and so forth, we can also find the steady state concentration of C_{s1} as a function of, among other things, the number of competing guide RNAs present. Zhang & Voigt do so using this formula:

$$C_{s1} = \frac{\alpha_1 C_{tot}}{\beta + \alpha_1 + N\alpha_x} \tag{3}$$

where α is a transcription rate of gRNA, C_{tot} is the concentration of the total dCas pool, $\beta = \frac{\delta_1}{K_1}$ is the dynamic pressure toward free gRNA (degradation δ_1 sucking away gRNA + dissociation $\frac{1}{K_1}$ forcing it off), and N is the number of competing, otherwise identical guides.

However, this equation actually tells us the concentration of dCas:gRNA *and* dCas:gRNA:promoter. This only holds as a measure of free dCas:gRNA in the regime of very small amounts of promoter, however – if there is a lot of promoter (relative to dCas), some C_{s1} will also bind away to the promoter. The extreme case is when there are bazillions of promoters and a few dCas molecules – all of the dCas will be bound up, and there will be none free.

To correct this, we just have to subtract off promoter-bound dCas:

$$C_{s1} = \frac{\alpha_1 C_{tot}}{\beta + \alpha_1 + N\alpha_x} - P_C \tag{4}$$

To find the steady-state concentation of P_C, we use 1) flux-balance between bound and free promoter and 2) mass conservation of promoter:

$$P_C = \frac{k_f}{k_r} C_S P \tag{5}$$

$$P_{tot} = P_C + P \tag{6}$$

Combining these and rearranging to solve for P_C, we get

$$P_C = \frac{K C_S P_{tot}}{1 + K C_S} \tag{7}$$

where, again, K is the *association* constant for binding between dCas:gRNA and promoter. Substituting this into (**??**) and rearranging yields a quadratic polynomial in C_S:

$$0 = -K C_S^2 + C_S \left(K C^* - K P_{tot} - 1\right) + C^* \tag{8}$$

where $C^* = \frac{\alpha_1 C_{tot}}{\beta+\alpha_1+N\alpha_X}$. This has an analytic solution, but it isn't very useful to look at; we'll be simply asking numpy to find the positive root for us.

We can use a similar analysis to find the fold-change caused by an activating dCas:gRNA. The only differences are that 1) neither $r_f \approx 0$ nor $r_C \approx 0$ and 2) the fold-change is (bound transcription/unbound transcription) instead of (unbound transcription/bound transcription).

$$AFC = \frac{r_f P + r_C K C_{s1} P}{r_f (P + K C_{s1} P)} = \frac{1 + \frac{r_C}{r_f} K C_{s1}}{1 + K C_{s1}}$$

With some parameters, we can now directly calculate fold-changes for activators and repressors under various conditions. Zhang and Voigt estimate most of these parameters by fitting to some of their data. My own modeling makes me question the accuracy (identifiability?) of those fits, but they'll at least generate curves that match real data:

- $\beta = 3.0 \times 10^{-2}$ nM^{-1}s^{-2}
- $\alpha_1 = 7.6 \times 10^{-3}$ nM/s
- $\alpha_x = 2.3 \times 10^{-2}$ nM/s
- $K = 2.9$ nM^{-1}s^{-1}

Curiously, they don't provide a fit to C_{tot}, presumably because it wasn't identifiable and just scales the y-axis. I'm going to make a blind guess at $C_{tot} = 100$ nM.

Finally, we need the activation ratio $\frac{r_C}{r_f}$ of the activator. Based on Dong & Zalatan, I think a good best-case estimate of the activation ratio, for an optimized SoxS activator, is around $\frac{r_C}{r_f} = r_A = 50$.

With these parameters, we can plot out $RFC(N)$ and $AFC(N)$:

When the repressor isn't present, this simplifies to $r = r_f P$; therefore, the fold-change caused by the repressor is:

$$RFC = \frac{r_f P_{tot}}{r_f P} = \frac{r_f (P + K C_{s1} P)}{r_f P} = 1 + K C_{s1} \tag{9}$$

Using some steady-state assumptions and dynamics of C and C_{s1} and s_1 and so forth, we can also find the steady state concentration of C_{s1} as a function of, among other things, the number of competing guide RNAs present. Zhang & Voigt do so using this formula:

$$C_{s1} = \frac{\alpha_1 C_{tot}}{\beta + \alpha_1 + N\alpha_x} \tag{10}$$

where α is a transcription rate of gRNA, C_{tot} is the concentration of the total dCas pool, $\beta = \frac{\delta_1}{K_1}$ is the dynamic pressure toward free gRNA (degradation δ_1 sucking away gRNA + dissociation $\frac{1}{K_1}$ forcing it off), and N is the number of competing, otherwise identical guides.

However, this equation actually tells us the concentration of dCas:gRNA *and* dCas:gRNA:promoter. This only holds as a measure of free dCas:gRNA in the regime of very small amounts of promoter, however – if there is a lot of promoter (relative to dCas), some C_{s1} will also bind away to the promoter. The extreme case is when there are bazillions of promoters and a few dCas molecules – all of the dCas will be bound up, and there will be none free.

To correct this, we just have to subtract off promoter-bound dCas:

$$C_{s1} = \frac{\alpha_1 C_{tot}}{\beta + \alpha_1 + N\alpha_x} - P_C \tag{11}$$

To find the steady-state concentation of P_C, we use 1) flux-balance between bound and free promoter and 2) mass conservation of promoter:

$$P_C = \frac{k_f}{k_r} C_S P \tag{12}$$

$$P_{tot} = P_C + P \tag{13}$$

Combining these and rearranging to solve for P_C, we get

$$P_C = \frac{K C_S P_{tot}}{1 + K C_S} \tag{14}$$

where, again, K is the *association* constant for binding between dCas:gRNA and promoter. Substituting this into (**??**) and rearranging yields a quadratic polynomial in C_S:

$$0 = -K C_S^2 + C_S \left(K C^* - K P_{tot} - 1 \right) + C^* \tag{15}$$

where $C^* = \frac{\alpha_1 C_{tot}}{\beta + \alpha_1 + N\alpha_X}$. This has an analytic solution, but it isn't very useful to look at; I'll be simply asking numpy to find the positive root for me.

We can use a similar analysis to find the fold-change caused by an activating dCas:gRNA. The only differences are that 1) neither $r_f \approx 0$ nor $r_C \approx 0$ and 2) the fold-change is (bound transcription/unbound transcription) instead of (unbound transcription/bound transcription).

$$AFC = \frac{r_f P + r_C K C_{s1} P}{r_f (P + K C_{s1} P)} = \frac{1 + \frac{r_C}{r_f} K C_{s1}}{1 + K C_{s1}}$$

With this, we can calculate RFC and AFC. We'll be defining two functions to calculate RFC – one for a perfect repressor (`RFC`) and one for a repressor with finite fold-repression (`imperfect_RFC`).

```
In [4]: def eq_Cs1(N, params):
            if isinstance(N, np.ndarray) or isinstance(N, list):
                return np.array([eq_Cs1(n, params) for n in N])
            K, a_1, C_tot, beta, a_x, r_A, P_tot = params
            frac_free = a_1 * C_tot / (beta + a_1 + N*a_x)
            c1 = -K
            c2 = K * frac_free - (K*P_tot + 1)
            c3 = frac_free

            coeffs = np.array([c1, c2, c3])
            roots = np.roots(coeffs)
            #print("N: %d; roots: (%f, %f)" % (N, roots[0], roots[1]))
            if roots[0] > 0 and roots[1] > 0:
                raise ValueError("Unexpectedly found two positive solutions for Cs1
            return roots[0] if roots[0] > 0 else roots[1]

        def RFC(N, params):
            K, a_1, C_tot, beta, a_x, r_A, P_tot = params
            C_s1 = eq_Cs1(N, params)
```

```
    return 1 + K * C_s1

def imperfect_RFC(N, params):
    # Cheat by calculating the AFC for a system with
    # "activation" equal to 1/r_A.
    repressor_params = copy.copy(params)
    repressor_params[5] = 1 / params[5]

    return 1 / AFC(N, repressor_params)

def AFC(N, params):
    K, a_1, C_tot, beta, a_x, r_A, P_tot = params
    C_s1 = eq_Cs1(N, params)

    return (1 + r_A * K * C_s1) / (1 + K * C_s1)
```

4 Parameters

With some parameters, we can now directly calculate fold-changes for activators and repressors under various conditions. Zhang and Voigt estimate most of these parameters by fitting to some of their data:

- $\beta = 3.0 \times 10^{-2}\ \text{nM}^{-1}\text{s}^{-2}$
- $\alpha_1 = 7.6 \times 10^{-3}\ \text{nM/s}$
- $\alpha_x = 2.3 \times 10^{-2}\ \text{nM/s}$
- $K = 2.9\ \text{nM}^{-1}\text{s}^{-1}$

Curiously, they don't provide a fit to C_{tot}, presumably because it wasn't identifiable and just scales the y-axis. I'm going to make a blind guess at $C_{tot} = 100$ nM.

Finally, we need the activation ratio $\frac{r_C}{r_f}$ of the activator. Based on Dong & Zalatan, I think a good best-case estimate of the activation ratio, for an optimized SoxS activator, is around $\frac{r_C}{r_f} = r_A = 50$.

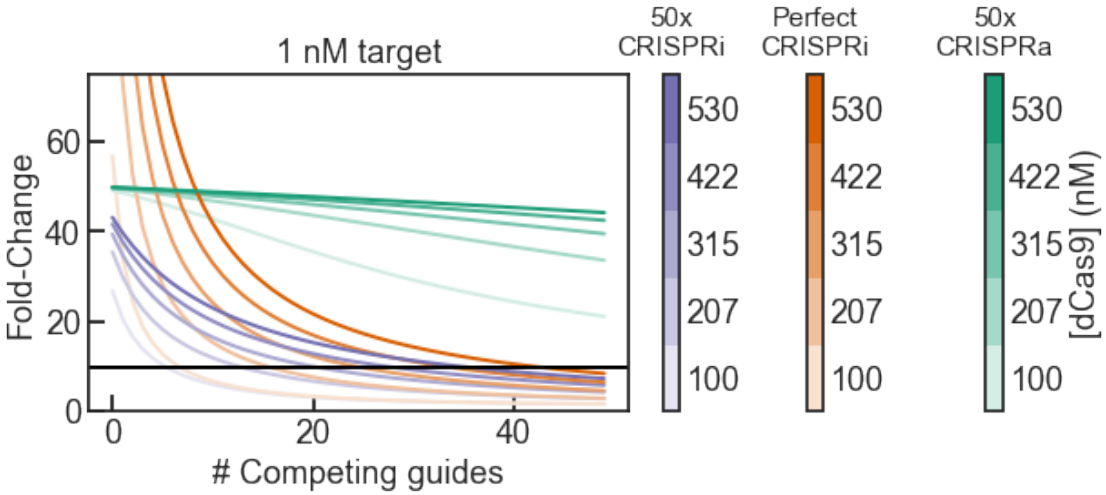
```
In [5]: # All parameters in units of nM/s except C_tot (nM), K_1 (nM), and r_A (uni
        K     = 2.9
        a_1   = 7.6e-3
        C_tot = 100
        beta  = 3.0e-2
        a_x   = 2.3e-2
        r_A   = 50
```

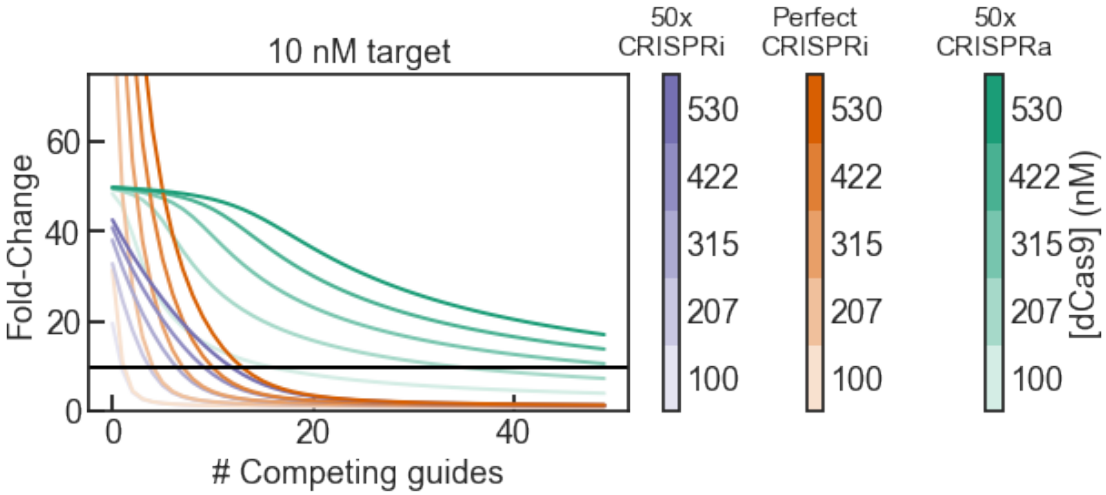
5 Figure 2: Scaling of CRISPRi and CRISPRa, as fold changes

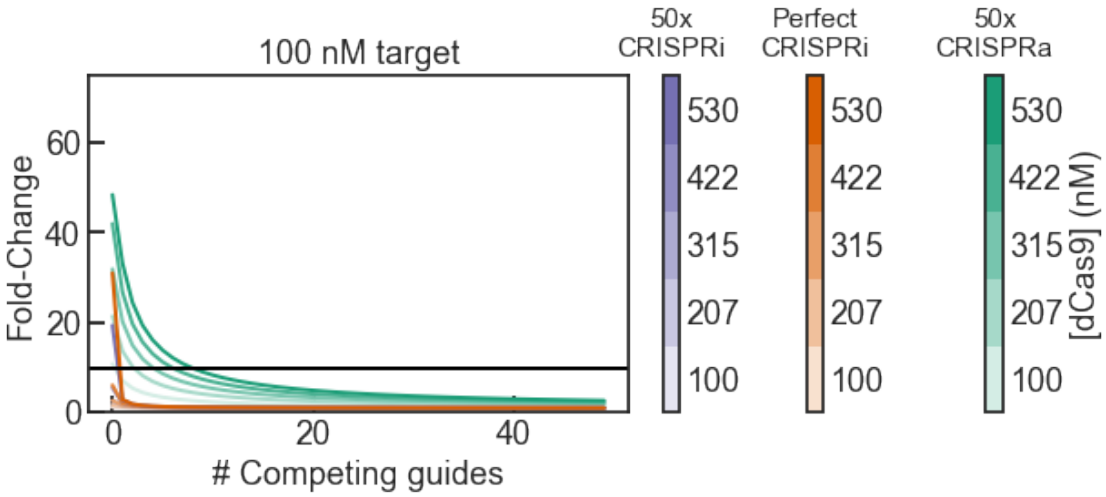
Let's calculate out $RFC(N)$ and $AFC(N)$. In each plot below, each line tracks how the fold change of the system changes as additional off-target gRNAs are added. Different shades of line represent systems with different concentrations of dCas9.

In [6]:
```
Cs = np.linspace(100,530,5) # dCas9 concentrations
Ns = np.array(range(50)) # Numbers of competing gRNAs

# Define discrete colormaps to use with these dCas9 concentrations.
# Will use these colors to show dCas9 concentrations.
color_idxs = np.linspace(0.1, 1, len(Cs))
orange_colormap = discretize_fade_colormap(orange_fade_colormap, min(Cs)/ma
                                           1.0, len(Cs))
green_colormap = discretize_fade_colormap(green_fade_colormap, min(Cs)/max
                                           1.0, len(Cs))
purple_colormap = discretize_fade_colormap(purple_fade_colormap, min(Cs)/ma
                                           1.0, len(Cs))

for P_tot in [1, 10, 100]:
    params = [K, a_1, C_tot, beta, a_x, r_A, P_tot]
    plt.figure(figsize=(9,4))
    for i, C in enumerate(Cs):
        params[2] = C

        plt.plot(Ns, imperfect_RFC(Ns, params), color = purple_colormap(col
                 label = "Repressor FC, imperfect")
        plt.plot(Ns, RFC(Ns, params), color = orange_colormap(color_idxs[i]
                 label = "Repressor FC")
        plt.plot(Ns, AFC(Ns, params), color = green_colormap(color_idxs[i])
                 label = "Activator FC")

    # Labeling
    plt.xlabel("# Competing guides")
    plt.ylabel("Fold-Change")
    plt.title("%d nM target" % (P_tot))

    # Make a colorbar, make it nice.
    green_sm = plt.cm.ScalarMappable(cmap=green_colormap,
                                     norm=plt.Normalize(vmin=Cs[0], vmax=Cs
    orange_sm = plt.cm.ScalarMappable(cmap=orange_colormap,
                                      norm=plt.Normalize(vmin=Cs[0], vmax=Cs
    purple_sm = plt.cm.ScalarMappable(cmap=purple_colormap,
                                      norm=plt.Normalize(vmin=Cs[0], vmax=C
    green_cbar  = plt.colorbar(green_sm)
    orange_cbar = plt.colorbar(orange_sm)
    purple_cbar = plt.colorbar(purple_sm)
    green_cbar.ax.set_title("50x\nCRISPRa", size = 15, y = 1.02)
    orange_cbar.ax.set_title("Perfect\nCRISPRi", size = 15, y = 1.02)
    purple_cbar.ax.set_title("50x\nCRISPRi", size = 15, y = 1.02)
    green_cbar.set_label(r'[dCas9] (nM)')

    c_int = (max(Cs) - min(Cs)) / (len(Cs))
    C_midpoints = [c_int*i + c_int/2 + min(Cs) for i in range(len(Cs))]
```

```
C_labels = [str(int(c)) for c in Cs]
green_cbar.set_ticks(C_midpoints)
green_cbar.set_ticklabels(C_labels)
orange_cbar.set_ticks(C_midpoints)
orange_cbar.set_ticklabels(C_labels)
purple_cbar.set_ticks(C_midpoints)
purple_cbar.set_ticklabels(C_labels)

plt.axhline(10, color = "black")
plt.ylim(0, 75)
plt.tight_layout()
plt.show()
```







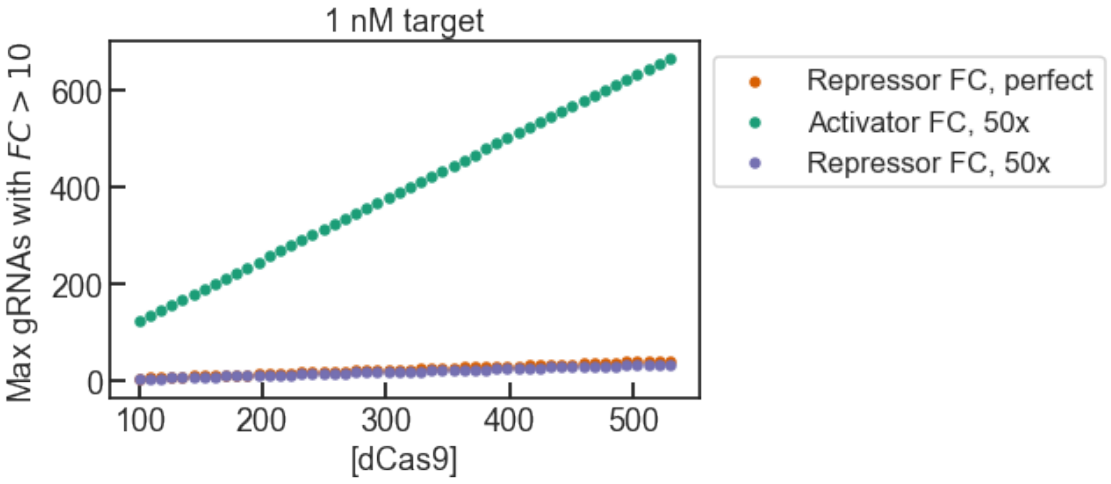
6 Figure 3: Scaling of CRISPRi and CRISPRa, as "maximum number of gRNAs"

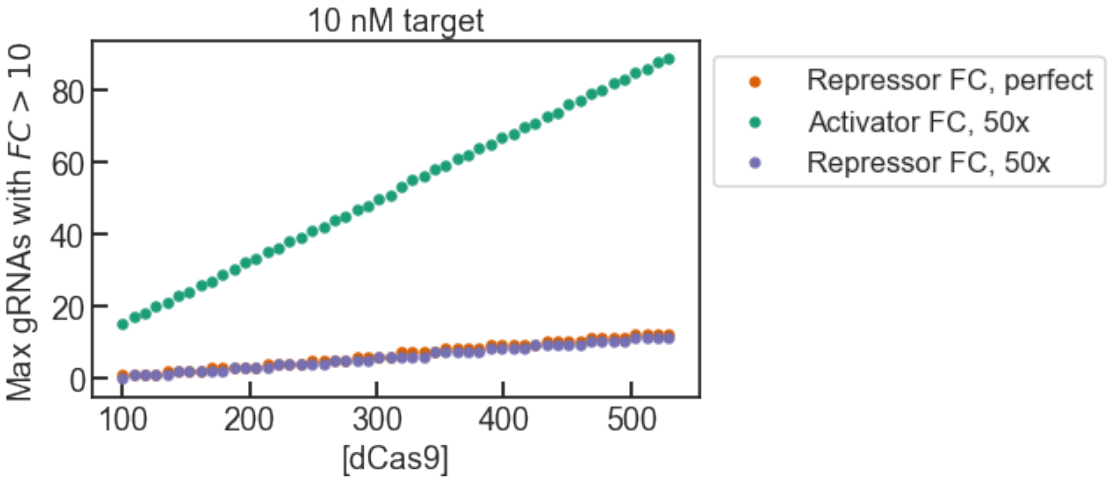
We can express the same data as a "capacity" – what we really want to know is how many gRNAs can be added to a system without messing up the original gRNA. We will arbitrarily (but, we hope, reasonably) define a 10x repression or activation as sufficiently "not messed up" to use. How many non-targeting gRNAs can each system support, under this definition, before the on-target gRNA dips below that 10-fold threshold?

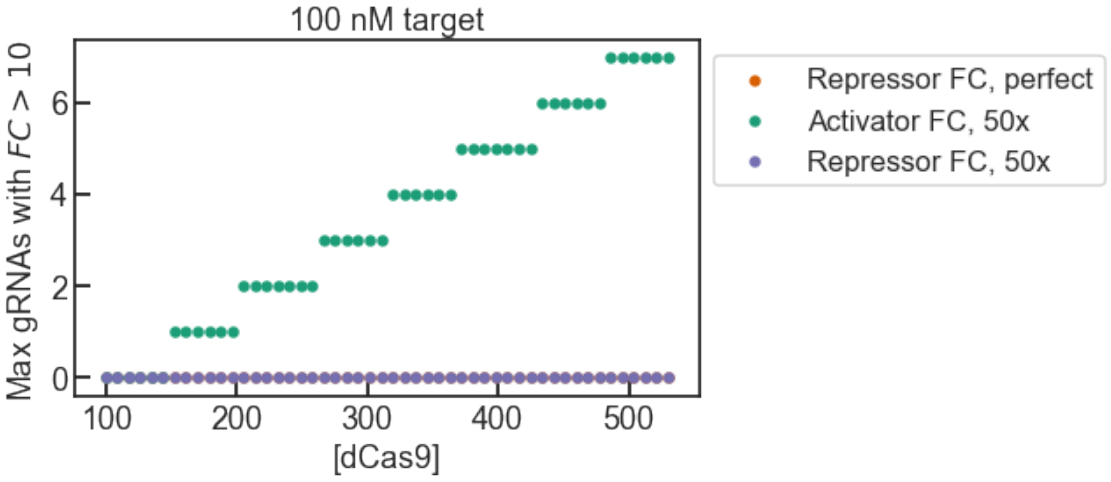
```
In [7]: def max_N_above_10_fold(params, performance_func):
            '''
            Returns the maximum number of gRNAs for which
            performance_func > 0, with parameters given by
            params. performance_func must be a function whose
            first argument is a number of competing gRNAs and
            whose second argument is a set of parameters.
            '''
            N = 1
            while True:
                if performance_func(N, params) > 10:
                    N *= 2
                else:
                    break
            upper_bound = N
            lower_bound = N//2
            while upper_bound - lower_bound > 10:
                midpoint = int((upper_bound + lower_bound)//2)
                if performance_func(midpoint, params) > 10:
                    lower_bound = midpoint
```

```
            else:
                upper_bound = midpoint
        for N in range(lower_bound, upper_bound+1):
            if performance_func(N, params) < 10:
                return max(0, N-1)

    def max_N_with_varied_param(param_range, param_idx, params, performance_fun
        '''
        Applies max_N_above_10_fold to a list of parameter sets in
        which one of the parameters (the one at index param_idx) varies
        over param_range.
        '''
        param_sets = np.zeros((len(param_range), len(params)))
        for i, param_val in enumerate(param_range):
            param_sets[i,:] = params
            param_sets[i,param_idx] = param_val
        return np.array([max_N_above_10_fold(param_sets[i,:], performance_func)
                         for i in range(len(param_range))])
```

In [8]:
```
Cs = np.linspace(100,530,50)

for P_tot in [1, 10, 100]:
    params = [K, a_1, C_tot, beta, a_x, r_A, P_tot]

    plt.figure(figsize=(9,4))

    plt.scatter(Cs, max_N_with_varied_param(Cs, 2, params, RFC),
                color = orange_fade_colormap(color_idxs[-1]),
                label = "Repressor FC, perfect")
    plt.scatter(Cs, max_N_with_varied_param(Cs, 2, params, AFC),
                color = green_fade_colormap(color_idxs[-1]),
                label = "Activator FC, 50x")
    plt.scatter(Cs, max_N_with_varied_param(Cs, 2, params, imperfect_RFC),
                color = purple_fade_colormap(color_idxs[-1]),
                label = "Repressor FC, 50x")

    plt.xlabel("[dCas9]")
    plt.ylabel(r"Max gRNAs with $FC>10$")
    plt.title("%d nM target" % (P_tot))
    plt.legend(bbox_to_anchor=[1,1])
    plt.tight_layout()
    plt.show()
```


1 nM target
Max gRNAs with FC > 10
600
400
200
0
100
200
300
400
500
[dCas9]
Repressor FC, perfect
Activator FC, 50x
Repressor FC, 50x


10 nM target
Max gRNAs with FC > 10
80
60
40
20
0
100
200
300
400
500
[dCas9]
Repressor FC, perfect
Activator FC, 50x
Repressor FC, 50x


100 nM target
Max gRNAs with FC > 10
6
4
2
0
100
200
300
400
500
[dCas9]
Repressor FC, perfect
Activator FC, 50x
Repressor FC, 50x

7 Figure 4: Figure 3, but with randomized parameters

To ensure our results aren't dependent on our exact choice of parameters, we can re-make the same kinds of plots as in Figure 3, but sampling parameters from some distribution around our best estimates.

Specifically, we will sample as follows: * P_{tot}: Fixed at 10 nM. We have already shown how target promoter concentrations affect CRISPRi/a scaling; no great need to change that here. * C_{tot}: Varies as in previous figures. * $\alpha_x = \alpha_1 * \frac{2.3\times10^{-2}}{7.6\times10^{-3}}$: Changing α_x (with respect to α_1 just adds more competing gRNA – it's equivalent to changing N. Since we'll already be varying N on one axis, changing $\frac{\alpha_X}{\alpha_1}$ would functionally just add noise by transforming the `[gRNA]`-axis for each sample. Therefore, we're going to pin the ratio $\frac{\alpha_x}{\alpha_1}$ to a constant (the one in our best-guess parameter set). Changes in this ratio are equivalent to stretching or compressing the `[gRNA]`-axis. * $r_A \sim N(50, 10)$: We estimate fold-activation of the CRISPRa activator from data figures in [2]. This particular parameter, unlike most of the parameters in this model, is quite straightforward to estimate – it's roughly the ratio of flurescences of activated cells and unactivated cells, modulo some background subtraction. Respecting this high certainty, we sample this parameter from a normal distribution around a best-guess value of 50, with a relatively low standard deviation. * All other parameters will be sampled from a log-normal distribution around their best-estimate values, as determined in [1], with a $\log_{10}$ standard deviation of 0.5. This *very roughly* represents an undertainty in the order of magnitude of the parameter with a "50% confidence range" (informally speaking) of ~$\frac{1}{3}$x to 3x the best-guess value.

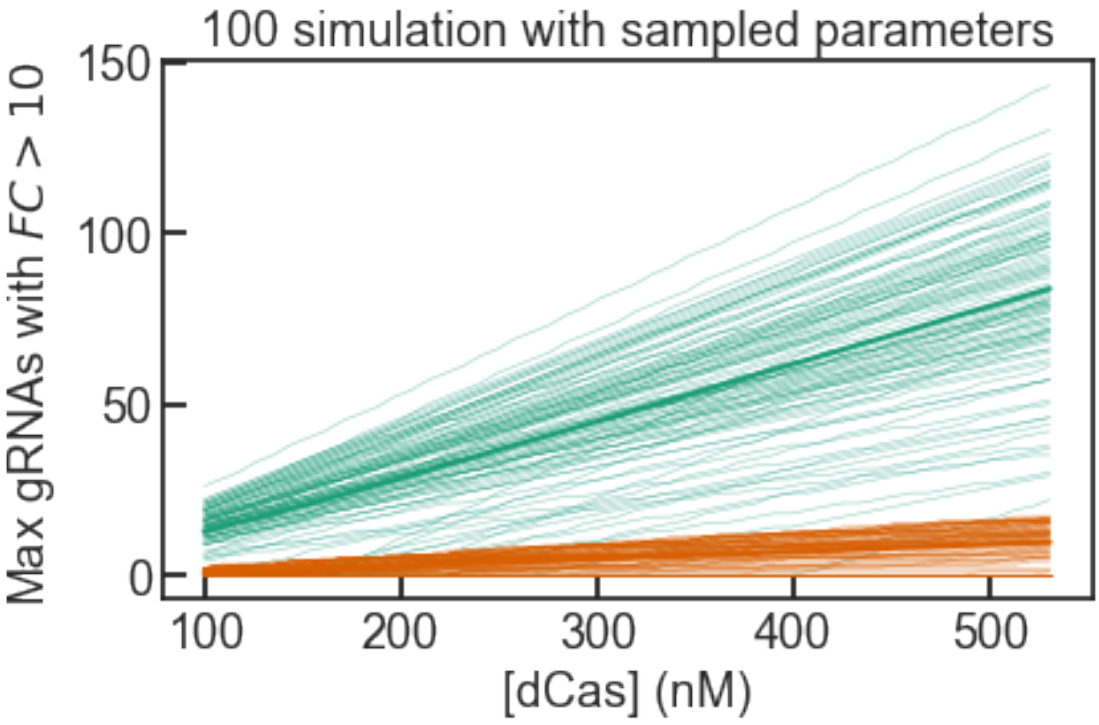
```
In [9]: def sampled_param_set(params):
            K, a_1, C_tot, beta, a_x, r_A, P_tot = params

            SD_multiple = 0.2

            K_sample     = K * 10**np.random.normal(0, 0.5)
            a_1_sample   = a_1 * 10**np.random.normal(0, 0.5)
            C_tot_sample = C_tot * 10**np.random.normal(0, 0.5)
            beta_sample  = beta * 10**np.random.normal(0, 0.5)
            a_x_sample   = a_1_sample * a_x / a_1 # np.random.normal(a_x, a_x*SD_mu
            r_A_sample   = np.random.normal(r_A, r_A*0.2)
            P_tot_sample = 10

            return np.array([K_sample, a_1_sample, C_tot_sample,
                             beta_sample, a_x_sample, r_A_sample,
                             P_tot_sample])
```

```
In [10]: ########################################
         # CALCULATE RFC/AFC FOR SAMPLED PARAMS #
         ########################################

         # Change this to change the number of parameter sets sampled. In the paper
         # n_trials = 1000. Set low here so it will run quickly.
```

```
n_trials = 100

# Other setup.
P_tot = 10
Cs = np.linspace(100,530,50)
color_idxs = np.linspace(0.1, 1, len(Cs))
param_names = [r"$K$", r"$a_1$", r"$C_{tot}$",
               r"$\beta$", r"$a_x$", r"$r_A$",
               r"$P_{tot}$"]
AFCs = -1*np.ones((n_trials, len(Cs)))
RFCs = -1*np.ones((n_trials, len(Cs)))

params = [K, a_1, C_tot, beta, a_x, r_A, P_tot]

for i in range(n_trials):
    param_sample = sampled_param_set(params)
    AFCs[i,:] = max_N_with_varied_param(Cs, 2, param_sample, AFC)
    RFCs[i,:] = max_N_with_varied_param(Cs, 2, param_sample, RFC)
```

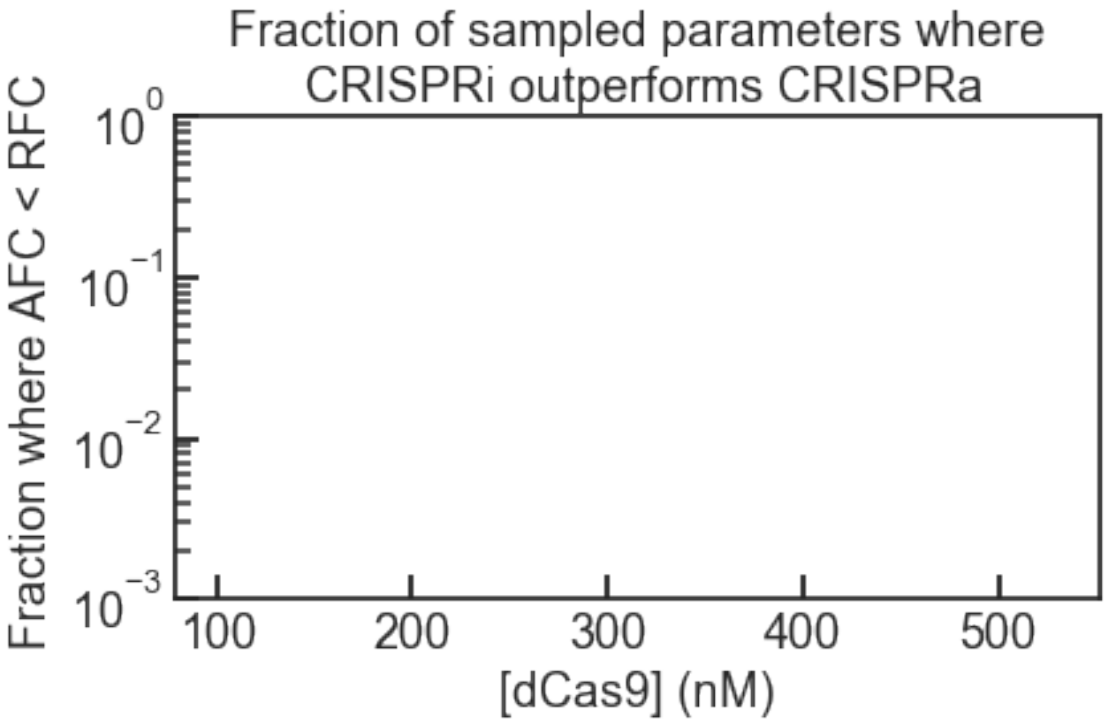
Plot the fold-change curves for these simulations (Fig. 4a)

```
In [11]: ################
         # PLOT RESULTS #
         ################
         plt.figure(figsize=(6,4))
         plt.plot(Cs, AFCs.transpose(), color = green_fade_colormap(color_idxs[-1])
                  lw = 0.5, alpha = 0.5);
         plt.plot(Cs, RFCs.transpose(), color = orange_fade_colormap(color_idxs[-1]
                  lw = 0.5, alpha = 0.5);
         plt.plot(Cs, AFCs.mean(axis = 0).transpose(),
                  color = green_fade_colormap(color_idxs[-1]), lw = 2)
         plt.plot(Cs, RFCs.mean(axis = 0).transpose(),
                  color = orange_fade_colormap(color_idxs[-1]), lw = 2)

         plt.xlabel("[dCas] (nM)")
         plt.ylabel(r"Max gRNAs with $FC>10$")
         plt.title("%d simulation with sampled parameters" % n_trials)
         plt.tight_layout()
```



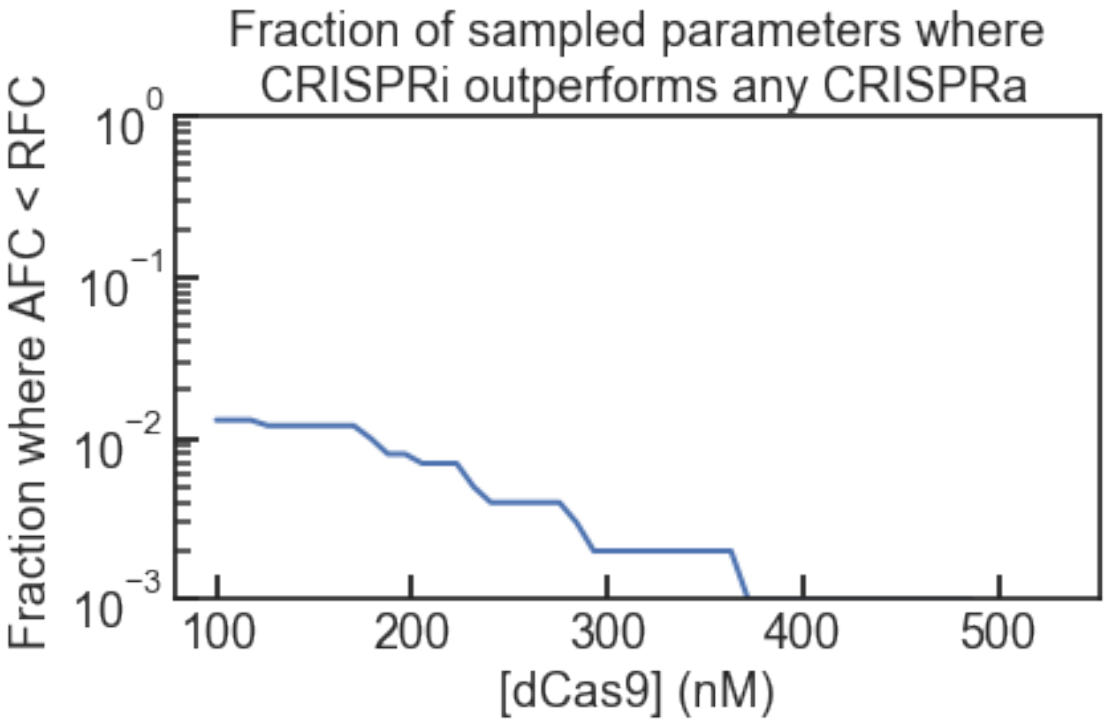
How often does the activator system work less well than the repressor, with the same parameters?

```
In [12]: max_RFC = max(RFCs[:,-1])
         n_worse_thans = [len([i for i in range(n_trials) if AFCs[i,j] < RFCs[i,j]]
                              for j in range(len(Cs))]
         worsts = [len([a for a in AFCs[:,i] if a < max(RFCs[:,i])]) for i in range

         plt.plot(Cs, [w/1000 for w in n_worse_thans])
         plt.xlabel("[dCas9] (nM)")
         plt.ylabel("Fraction where AFC < RFC")
         plt.title("Fraction of sampled parameters where\n CRISPRi outperforms CRIS
         plt.ylim(0.001, 1)
         plt.yscale("log")
         plt.tight_layout()
```



Answer: never. We can also check a more stringent criteria – how often does CRISPRa perform worse than *any* sampled CRISPRi system? (Fig. 4b).

```
In [13]: max_RFC = max(RFCs[:,-1])
         worsts = [len([a for a in AFCs[:,i] if a < max(RFCs[:,i])]) for i in range

         plt.plot(Cs, [w/1000 for w in worsts])
         plt.xlabel("[dCas9] (nM)")
         plt.ylabel("Fraction where AFC < RFC")
         plt.title("Fraction of sampled parameters where\n CRISPRi outperforms any
         plt.ylim(0.001, 1)
         plt.yscale("log")
         plt.tight_layout()
```



8 Sensitivity analysis:

We will also check the local sensitivity of our results to each parameter. To be precise, we'll define "fold change overperformance" as the ratio between the fold-change of activation for the activator and the fold change of repression for the repressor, for a single set of parameters and a single number of competing gRNAs:

```
In [14]: def fold_change_overperformance(N, params):
             return AFC(N, params) / RFC(N, params)
```

We'll calculate sensitivities of this measure by calculating its derivative with respect to each parameter, normalized to parameter scale and absolute overperformance at the best-guess parameter set:

$$\text{normalized sensitivity}(\text{params}) = \frac{\text{params}}{\text{performance}(\text{params})} * \text{sensitivity}(\text{params})$$

```
In [15]: N = 30
         P_tot = 10

         sens_params    = np.array([K, a_1, C_tot, beta, a_x, r_A, P_tot])
         gradient_sens = nd.Gradient(lambda p:fold_change_overperformance(N, p))(se
         norm_sens = gradient_sens * sens_params / fold_change_overperformance(N, s
         for i in range(len(norm_sens)):
             print(r"Normalized sensitivity to %s: %f" % (param_names[i], norm_sens
```

```
Normalized sensitivity to $K$: 0.026457
Normalized sensitivity to $a_1$: 0.709052
Normalized sensitivity to $C_{tot}$: 0.716537
Normalized sensitivity to $\beta$: -0.029544
Normalized sensitivity to $a_x$: -0.679508
Normalized sensitivity to $r_A$: 0.848310
Normalized sensitivity to $P_{tot}$: -0.690079
```

Another performance measure we can use is the "scaling overperformance", which measures how many more gRNAs a CRISPRa system can support than a CRISPRi system with the same parameters, while staying above 10x activation/repression by the targeting gRNA (like in Figure 3).

Calculating sensitivity to this measurement requires cheating and allowing real values of N – gradient analysis only works on continuous functions.

In [16]:
```
def continuous_performance(params, func):
    N_guess = max_N_above_10_fold(params, func)
    if N_guess <= 0:
        return 0
    N = scipy.optimize.root_scalar(lambda N: func(N, params) - 10, method
                                   bracket = (N_guess-1, N_guess+1), x0 =
    return N.root

def scaling_overperformance(params, imperfect = False):
    temp_params = copy.copy(params)
    temp_params[4] = temp_params[1]
    N_activation = continuous_performance(params, AFC)
    if imperfect:
        N_repression = continuous_performance(params, imperfect_RFC)
    else:
        N_repression = continuous_performance(params, RFC)
    return N_activation / N_repression if N_repression > 0 else np.inf
```

In [17]:
```
P_tot = 30
C_tot = 530

sens_params   = np.array([K, a_1, C_tot, beta, a_x, r_A, P_tot])
scaling_gradient_sens = nd.Gradient(lambda p:scaling_overperformance(p))(s
scaling_norm_sens = scaling_gradient_sens * sens_params / scaling_overperf
for i in range(len(scaling_norm_sens)):
    print("\tNormalized scaling overperformance sensitivity to %s: %f" %
```

```
Normalized scaling overperformance sensitivity to $K$: -0.128736
Normalized scaling overperformance sensitivity to $a_1$: -0.267926
Normalized scaling overperformance sensitivity to $C_{tot}$: -0.335800
Normalized scaling overperformance sensitivity to $\beta$: 0.267926
Normalized scaling overperformance sensitivity to $a_x$: -0.000000
Normalized scaling overperformance sensitivity to $r_A$: 1.079925
```

```
        Normalized scaling overperformance sensitivity to $P_{tot}$: 0.207065


/Users/sclamons/anaconda/lib/python3.6/site-packages/numdifftools/extrapolation.py:
  converged = err <= tol
/Users/sclamons/anaconda/lib/python3.6/site-packages/numdifftools/extrapolation.py:
  old_sequence[-m + 1:]) * fact)
/Users/sclamons/anaconda/lib/python3.6/site-packages/numdifftools/limits.py:173: Ru
  outliers = (((abs(der) < (a_median / trim_fact)) +
/Users/sclamons/anaconda/lib/python3.6/site-packages/numdifftools/limits.py:174: Ru
  (abs(der) > (a_median * trim_fact))) * (a_median > 1e-8) +
/Users/sclamons/anaconda/lib/python3.6/site-packages/numdifftools/limits.py:175: Ru
  ((der < p25 - 1.5 * iqr) + (p75 + 1.5 * iqr < der)))
```

9 Important references:

1) S. Zhang, C. A. Voigt, Engineered dCas9 with reduced toxicity in bac- teria: implications for genetic circuit design, Nucleic acids research 46 (2018) 11115–11125.

2) C. Dong, J. Fontana, A. Patel, J. M. Carothers, J. G. Zalatan, Syn- thetic CRISPR-Cas gene activators for transcriptional reprogramming in bacteria, Nature Communications 9 (2018).